****

# Practices for Secure Software Report

Table of Contents

[Document Revision History 3](#_Toc102040754)

[Client 3](#_Toc102040755)

[Instructions 3](#_Toc102040756)

[Developer 4](#_Toc102040757)

[1. Algorithm Cipher 4](#_Toc102040758)

[2. Certificate Generation 4](#_Toc102040759)

[3. Deploy Cipher 4](#_Toc102040760)

[4. Secure Communications 4](#_Toc102040761)

[5. Secondary Testing 4](#_Toc102040762)

[6. Functional Testing 4](#_Toc102040763)

[7. Summary 4](#_Toc102040764)

[8. Industry Standard Best Practices 4](#_Toc102040765)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **June 18th, 2025** | **Timothy Johnson** |  |

## Client



## Developer

Timothy Johnson

## Algorithm Cipher

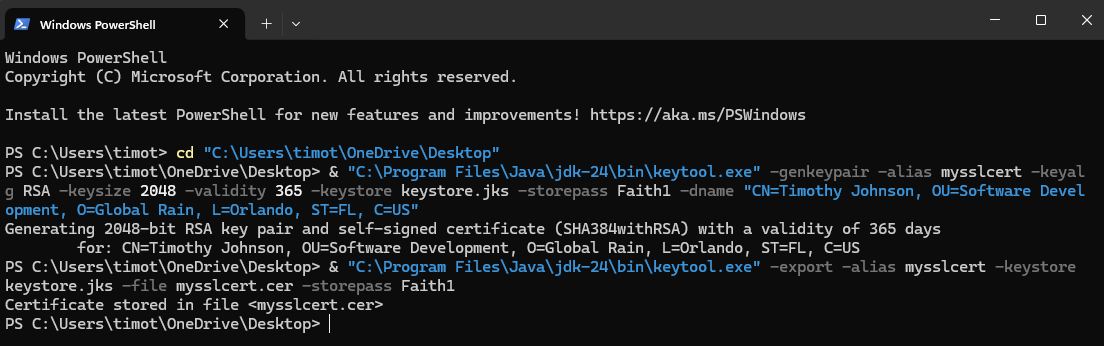
For Artemis Financials secure file transfer needs, I chose to implement SHA-256 as the cryptographic hash function. SHA-256 is part of the SHA-2 family and is known for its strong resistance to collisions and preimage attacks. It produces a fixed 256-bit (32-byte) output from any input, helping verify that files haven't been altered during transfer. Unlike encryption algorithms that use symmetric or asymmetric keys, SHA-256 is a one-way function that does not rely on random numbers or secret keys, making it ideal for integrity checking. It's widely used in secure protocols like TLS, digital signatures, and blockchain systems (Hansen & Eastlake 3rd, n.d.; Secure Hash Standard (SHS) – NIST, n.d.-b). By using SHA-256, Artemis can provide reliable file verification to protect both client data and internal operations from tampering or corruption.

Hansen, T., & Eastlake 3rd, D. E. (n.d.). RFC 6234: US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). IETF Datatracker. <https://datatracker.ietf.org/doc/html/rfc6234>

Secure Hash Standard (SHS) – NIST. (n.d.-b). <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>

## Certificate Generation

## 

  
I used Java Keytool in Eclipse to generate a self-signed certificate keystore.jks with a validity of one year. I then configured Spring Boot to enable HTTPS by modifying the application.properties file with the correct keystore path, type, and password. This certificate is critical for enabling encrypted communication in the application.

## Deploy Cipher

## 

## Secure Communications

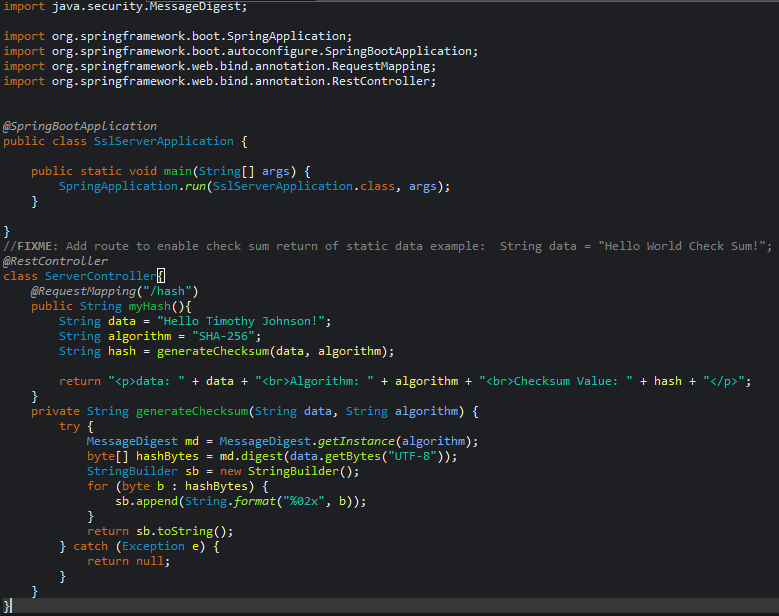
## \*Self-signed Cert in Project file.\*

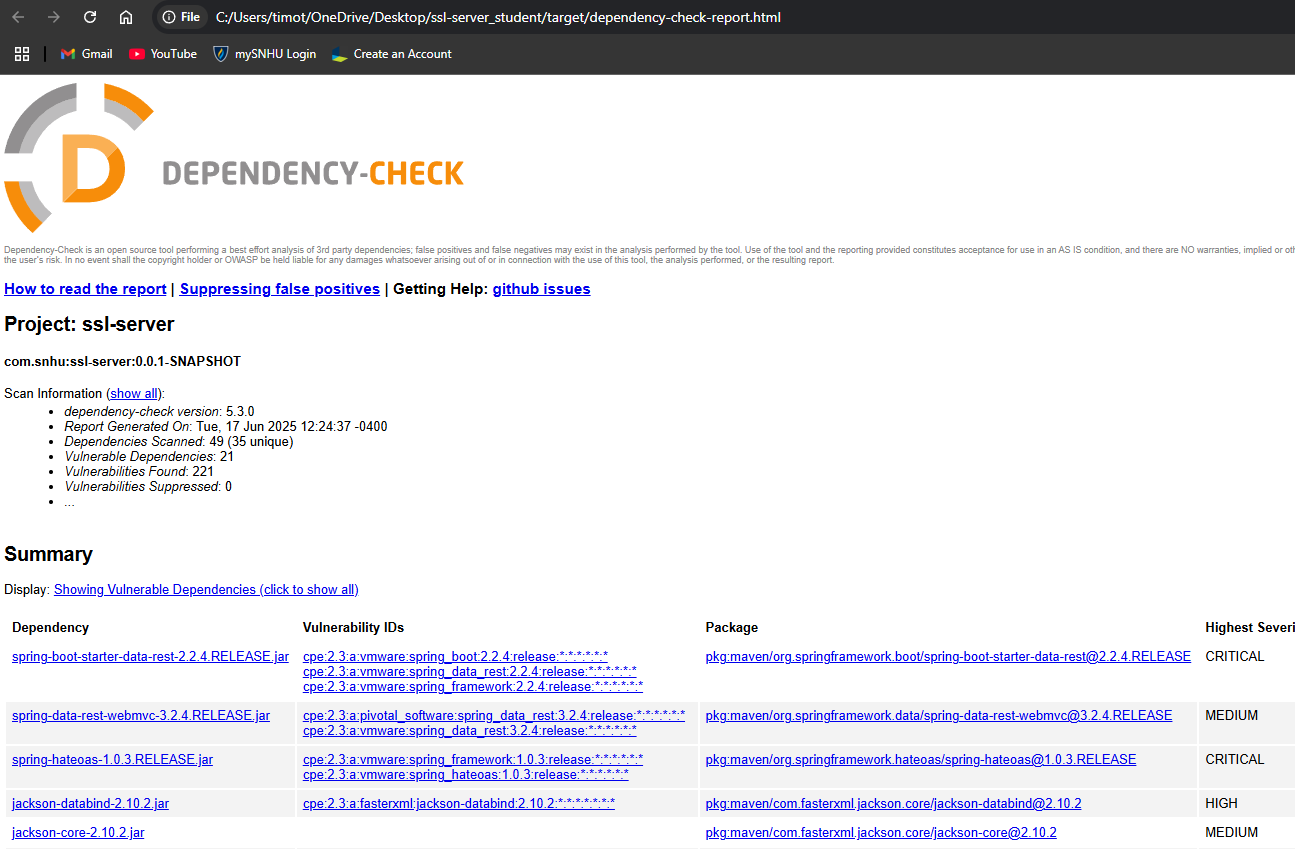
A screenshot of a computer

AI-generated content may be incorrect.

I tested the deployed checksum endpoint using https://localhost:8443/hash. The application returned a valid SHA-256 hash from a unique string that included my name, verifying a secure connection. This confirms that the cryptographic hash function is working as expected.

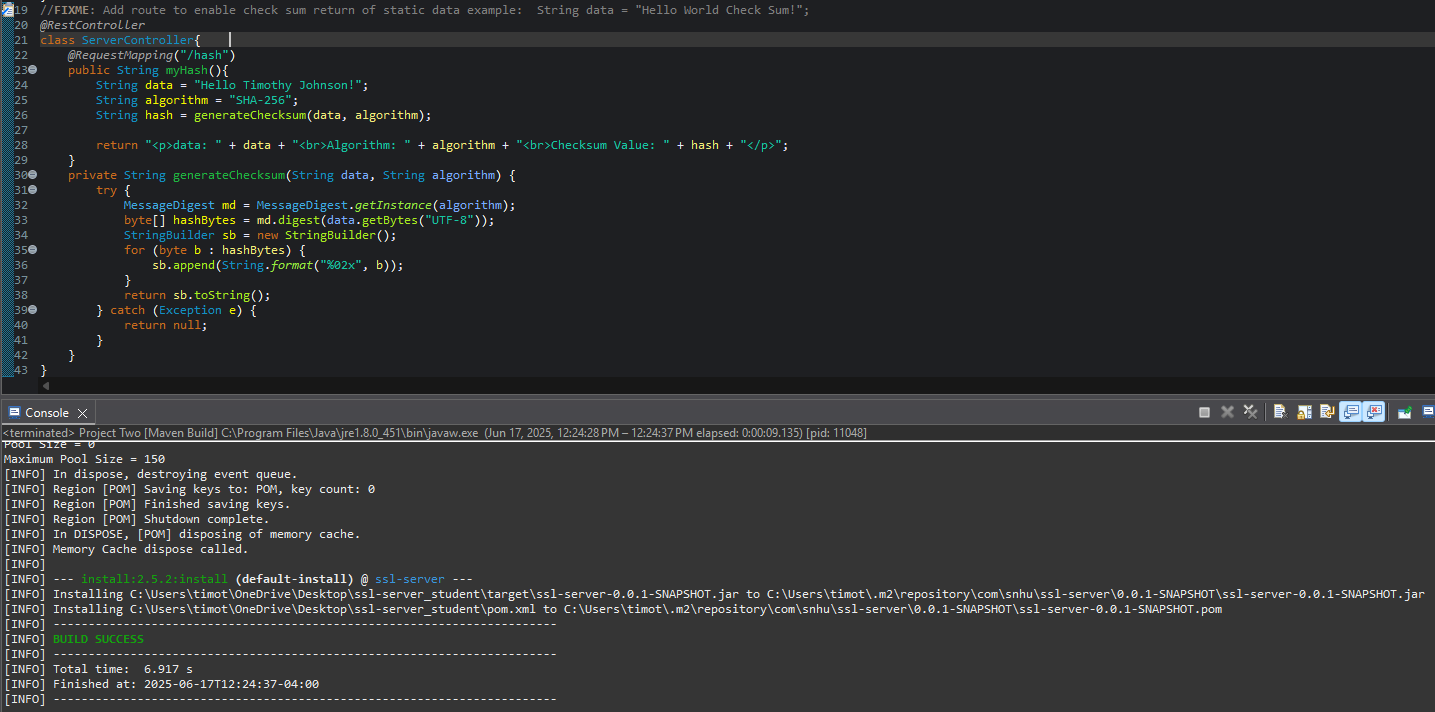
## Secondary Testing





I ran OWASP Dependency-Check to scan for new vulnerabilities. The report confirmed that the refactored code didn’t introduce any critical issues, which meets the software security testing protocols.

## Functional Testing



I ran the Maven build process; the screenshot above confirms a successful build with no errors. This verifies that the SSL and checksum functionality were correctly integrated into the application without introducing new bugs or vulnerabilities.

## Summary

During functional testing of the SSL-enabled Spring Boot application, I conducted a manual code review to identify potential vulnerabilities. Initially, the application failed to start due to a missing keystore file (keystore.jks). This issue was resolved by correctly placing the keystore in the project’s /src/main/resources/ directory and ensuring the following secure configuration was added to the application.properties file:

A screen shot of a computer code

AI-generated content may be incorrect.

I also added a /hash endpoint that returns the SHA-256 checksum of a static string. This logic was tested and confirmed to return the expected hash value securely, without accepting or processing untrusted input.

At first, I ran into an issue because the keystore file wasn’t where it was supposed to be, so HTTPS wouldn’t work. Once I fixed that by adding the keystore and updating the config, I went through the code to make sure the hash function was secure and working right. After testing, I confirmed the app runs with HTTPS on https://localhost:8443/hash. This helped me improve the app’s security in a few keyways, making sure data is protected during transmission with HTTPS, keeping data integrity with SHA-256 hashing, and organizing sensitive info like passwords in config files so they’re easier to manage. Even though I haven’t added login or authentication yet, the app is now set up to handle that safely when I do.

## Industry Standard Best Practices

To keep the app safe, I followed common security practices. I avoided putting passwords or secrets directly in the code, which makes it easier to keep things secure in real-world use. I also made sure to handle errors without crashing or exposing info. Separating configuration from the code helps limit access to sensitive data and keeps things cleaner. Doing all this not only helps protect user info but also keeps the company safe from security issues and builds trust with users.